基于COTS技术的高可靠通用容错计算机容错机制研究*)

欧中红[1,2] 袁由光[2] 赵晓勇[3]

(哈尔滨工程大学计算机科学与技术学院 哈尔滨 150001)[1]

(中船重工武汉709所 430074)[2] (三峡大学软件工程研究中心 宜昌 443000)[3]

摘 要 本文提出并实现了一种基于COTS部件、容错机制智能实现的、新颖的通用高可靠容错计算机系统。基于容错功能与用户应用相分离的原则,应用自主设计的智能管理模块,实现对COTS部件内部状态的可观察性。详细分析了系统的容错机制。利用提出的故障逃逸模型,分析了系统中的层次化故障检测和保护措施并估计了系统故障覆盖率。

关键词 智能容错,COTS部件,故障逃逸模型,故障覆盖率,容错开销

Research on Fault-tolerance Mechanism of COTS Components Based Generic Highly Reliable Fault-tolerant Computer

OU Zhong-Hong[1,2] YUAN You-Guang[2] ZHAO Xiao-Yong[3]

(Institute of Computer Science and Technology, Harbin Engineering University, Harbin 150001)[1]

(Wuhan 709th Institute of CSIC, Wuhan 430074)[2]

(Software Engineering Research Center, Three-Gorges University, Yichang 443000)[3]

Abstract Customized fault-tolerant computer is costly, hard to develop and has poor scalability. Based on COTS components and separation of fault-tolerance and applications, a kind of architecture of generic highly reliable cost-effective fault-tolerant computer is proposed and implemented. In order to increase the observability and controllability of the internal states of COTS, an intelligent fault-tolerance management module is devised.

Keywords Intelligent fault tolerance, COTS components, Fault escape model, Fault-tolerance coverage, Fault-tolerance cost

1 引言

为克服专门定制的高可靠容错计算机[1,2]容错计算的成本过高、可扩展能力弱、编程复杂,使用不方便以及升级能力差等缺陷,容错计算机设计人员开始研究低成本、扩展灵活、紧跟技术进步和通用的容错计算机,这种容错计算机以商用成熟硬件软件(Commercial off-the-Shelf,COTS)为基础。硬件技术和软件技术的进步使得这种可能性可以成为现实。美国NASA的JPL实验室等在航空航天应用领域[3~5]、欧洲面向多种应用领域都在研究和研制高性价比的容错计算机并取代专用容错计算机[6,7]。

但是,由于COTS硬件是面向大众化的市场需求,低成本的市场压力大,因此他们在设计时就很少考虑有关可靠性方面的需求。采用COTS技术来构造高可靠容错计算机有许多关键技术需要解决。

内部状态一致性问题:由COTS硬件构造高可靠容错计算机时一般为松散耦合的结构。传统专门定制的(customized)容错计算机采用紧耦合的方式来保证每一个冗余模块保持状态的一致性。要达到松散耦合的内部状态完全一致性是非常困难的。但是通过重新定义状态或者限制状态涵盖范围,辅助硬件和软件手段可以达到系统在输入/输出、内存关键区等状态的一致性。

容错开销(Fault tolerance cost)问题:容错是以牺牲系统一定的资源(包括时间资源、存储资源、计算资源等)为代价的。可以采用减少比较表决模块间状态次数的方法或者加快比较表决速度的办法来尽量减少容错开销。但是减少比较表决模块间状态次数的方法将会导致错误不能及时地检测出来,即错误潜伏期(Error Latency)长,导致错误的传播,受错误污染的区域扩大,系统恢复难度加大,时间延长。因此,容错开销需要与系统所要求的可靠性指标,故障模型,实时性指标等一起综合进行考虑。

故障覆盖率(Fault tolerance coverage)问题:衡量一个容错计算机性能的好坏,除可靠性、可用性指标外,故障覆盖率是一个更基本的指标(underlying)。故障覆盖率包括故障假设覆盖率(Fault assumption coverage)和故障/错误处理覆盖率(Error/fault handling coverage)[8]。故障假设覆盖包括失效模式覆盖和失效独立性覆盖。由于COTS部件本身的故障检测和处理机制很弱,需要采取额外的措施来提高系统的故障覆盖率。

容错透明性(Fault tolerance transparency)问题:容错计算对用户透明,用户和应用程序开发人员就像使用普通计算机一样使用该容错计算机。所以容错机制必须与应用分开。容错必须实现在硬件、设备驱动和操作系统以下。

本文提出并实现了一种新颖的、基于COTS部件、容错

*)本文受国防重点预先研究项目(413160201)资助。欧中红 博士研究生,主要研究方向:容错技术与可信计算,分布计算;袁由光 研究员,主要研究方向:计算机系统机构,可信计算。

机制智能实现的通用高可靠容错计算机系统(Intelligent COTS Components based Fault-tolerant Computer, ICFTC)。作者采用COTS技术设计了一种智能容错管理模块(Intelligent Fault-tolerance Management Module, IFMM)。基于容错功能与用户应用相分离的原则,应用该模块,可以实现对COTS部件(包括应用处理器模块的硬件状态和操作系统的运行状况等)内部状态的可观察性(Observability),通过对观察到的内部状态进行比较表来决检测故障,从而诊断故障、隔离故障恢复故障、进行系统重组等,以实现系统的高可靠性。该容错计算机支持DMR/TMR两种冗余方式,采用层次化的故障检测和保护措施来提高故障覆盖率,在容错方面该容错计算机具有较少的容错开销、较好的弹性、可扩展性和良好的透明性并支持在线维修。

2 一种智能的基于COTS部件的通用容错计算机(ICFTC)结构

2.1 硬件结构描述

ICFTC由二台(DMR)/三台(TMR)同构计算机(称节点C_1,C_2,C_3)组成,对用户而言是一台逻辑计算机。C_i主要采用高性能的商用部件,由高速专用网络(PHN)连接,而容错功能主要由智能容错管理模块(IFMM)实现。这种结构硬件实现相对简单,可以用标准模块、商用技术实现,如处理核心,网络接口完全可用成熟的商用标准模块;由于硬件和软件采用模块化结构技术,因此调试相对简单,故障容易定位;该系统为实现容错功能需要设置一系列的表决点,表决的次数可以根据任务需要灵活设置;易于升级,故障处理子系统可作为单独模块嵌入到标准硬件中,各功能子系统可以分别进行升级。

图1为ICFTC硬件结构图,图中PE_i($i=1,2,3$),IOM_i($i=1,2,3$)和$IFMM_i$($i=1,2,3$)组成三台物理计算机C_1、C_2、C_3。其中:

PE_i:处理单元,COTS部件,在容错服务器中每个PE包括CPU主机模块,每个模块均能带电热插拔。其中CPU模块自带ECC存储器,高性能的处理器,各C_i内的模块通过标准总线CPCI通信[9],其上的处理器称为应用处理器(host application processor)。

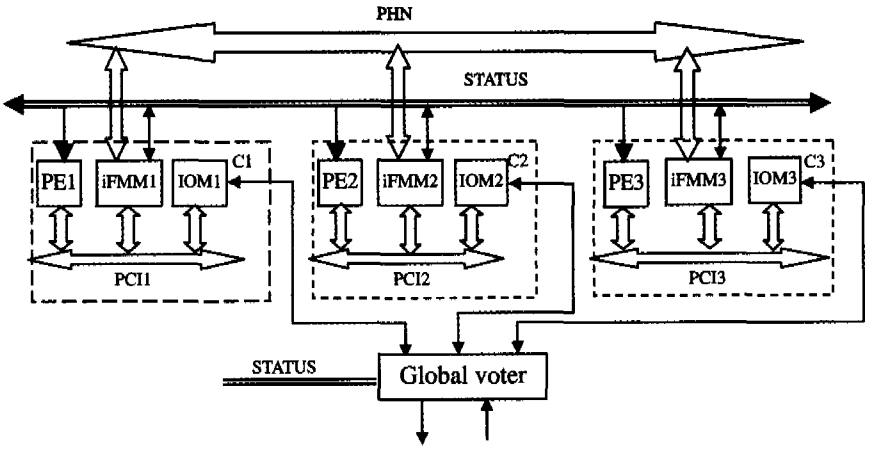


图1 ICFTC硬件体系结构

PHN(Proprietary High-speed Network):专用高速网,分别与C_1、C_2、C_3接口,为减少连线数量,采用高速串行总线。

$IFMM_i$(Intelligent Fault-tolerant Management Module):智能容错管理模块,通过一系列的算法实现C_i间同步、数据和状态表决、故障及错误检测、屏蔽、恢复等功能,通过标准总线CPCI总线与相应的处理单元PE_i通信。IFMM支持带电热插拔。

IOM_i:IO接口单元,如网络接口控制NIC,SCSI控制器等,COTS部件。

GLOBAL VOTER:通过接受$IFMM_i$模块输出的表决状态信息进一步确定故障节点,以防止故障节点向外输出错误结果,设计简单、可靠。

2.2 IFMM模块设计和内部状态的可观察性设计

IFMM是ICFTC的关键模块,由它实现大部分容错功能。图2是IFMM的硬件原理框图。可以看出IFMM本身自带CPU(称为容错用CPU),模块上还有内存、PCI总线接口。通过PCI总线接口,PE和IFMM模块上的内存可以相互共享。这就为PE内部状态的可观察提供了一条物理途径。图中可以看出IFMM模块上有PHN接口。PHN接口是双冗余的高速串行链路接口,传输速率可达125MB/S。

IFMM包括两个功能子模块:

通讯子模块:它接收PE_i($i=1,2,3$)送过来的表决数据,广播至C_j($j\neq i$),接收来自C_j($j\neq i$)表决数据。因此,一个待表决的数据对象正常情况下FMM内有三个数据拷贝,即一个是它自己的,另两个是C_j($j\neq i$)广播来的。这三个数据拷贝提交给下面的容错执行机构表决;

容错执行子模块:C_i间同步,表决,故障和错误检测、隔离及恢复。系统中待表决的数据对象包括表决点数据、应用数据和系统状态数据。

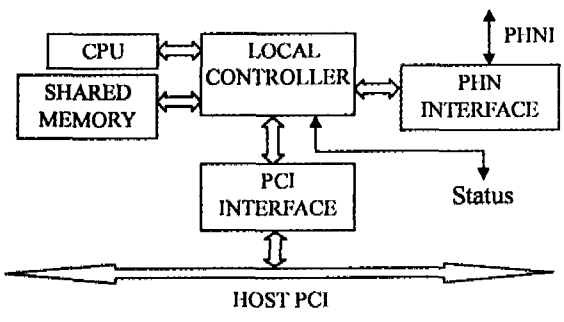


图2 IFMM模块硬件原理框图

表决的实现是通过专用的处理器根据一定的算法实现的，可以承担 PE 的表决负担，加快表决机构的执行，通过表决可以检测故障。节点中，每一个 C_i 都可认为是一个故障围堵区(fault containment region，FCR)，它以各自的软硬件为界，FCR 能够阻止发生在一个 FCR 内的故障传播到另一个 FCR，发生在一个 FCR 内的多故障可被视为单故障，因为别的 FCR 可以通过表决检测和屏蔽该故障。

各 C_i 中的 IFMM 组成系统的单一物理映像池 SSMP。组成 SSMP 的 IFMM 之间通过两条全双工的高速专用网络连接(PHN)，任何一个连接出错，IFMM 仍然相互通信。与传统的靠消息传递来交换信息从而达到 SC 的状态一致性的方法不同，本系统 IFMM 之间的网络连接用来实现 IFMM 之间的存储器映射。即 SSMP 实现全局存储器，系统的同步和错误检测通过存储器映像的访问得到，如图 3 所示。IFMM 等待、比较各 IFMM 送来的存储器映像。各 IFMM 与对应的 PE 是通过并行总线通信，IFMM 对 PE 而言是一个存储器设备。当 PE 需要同步时，它向局部的 IFMM 发出存储器访问，该访问被立刻映像到其他 IFMM，IFMM 在预先给定的期限内对内存映像进行处理，处理的结果通知 PE。

2.3 软件环境

ICFTC 采用开放源码的 LINUX 操作系统。通过对操作系统底层的某些针对性的修改，实现容错对用户的透明。

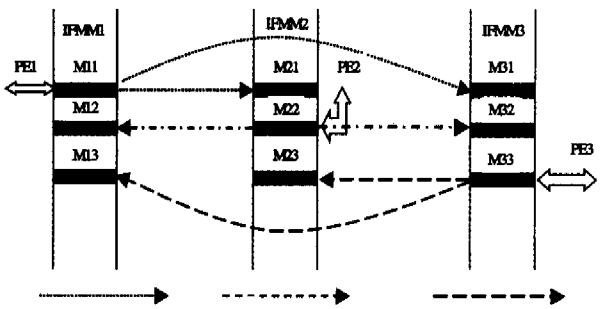


图 3 IFMM 间的存储器映射

2.4 容错机制研究

2.4.1 ICFTC 的计算模型

ICFTC 中：

$C_i=\{P_{ij}\mid j=0,1,\cdots\}$，$i=0,1,2$；$P$ 为进程。$\{P_{ij}\}$可以划分为两个子集合：操作系统的系统进程集$\{P_{ik}^s\}$(如定时进程、调度进程等)和用户应用进程集$\{P_{iq}^a\}$。每一个 P_{iq}^a 是由其输入集合$\{IN_{iq}^l\}$，和输出集合 $\{OUT_{iq}^m\}$组成。其中 $K_i=|\{P_{ik}^s\}|$为 C_i 内系统进程数，$Q_{iq}=|\{P_{iq}^a\}|$为 C_i 内应用进程数，$L_{iq}=|\{IN_{iq}^l\}|$为进程 P_{iq}^a 的输入点数，$M_{iq}=|\{OUT_{iq}^m\}|$为 P_{iq}^a 的输出点数，所以，进一步表示为：

$C_i=\{P_{ik}^s\}\cup\{P_{iq}^a\}$，$P_{iq}^a=\{OUT_{iq}^m\}\cup\{IN_{iq}^l\}$，$i=0,1,2$。

定义 1 输出表决点(Output Voting Point，OVP)：当应用进程准备向外输出数据 OUT_{iq}^m 时，它就达到一个表决点。此时，该进程被挂起等待表决结果。所有节点都达到该输出表决点且比较正确后才唤醒该被挂起进程继续执行。一个进程 P_{iq}^a 可以有多个输出表决点。

定义 2 输入一致性(Input Consensuses，IC)：输入一致性保证各复制进程都收到同一个输入。采用与 OVP 类似的方法取得输入一致性。

定义 3 内存关键区(Critical Part of memory，CP)：该区域位于 PE 内，是 PE 内存区域的一部分。中断向量表，进程表结构，内核代码所处的区域都为内存关键区。系统运行时频繁调用这些区域的数据和程序，一旦该区域出错，系统就会崩溃。在 IA-32(Intel Architecture-32)[10intel archi]结构中，当运行 LINUX 操作系统时，最低 1MB 空间是为操作系统内核保留的，所以也为内存关键区。运行期间保证各复制节点的内存关键区内容的一致性是非常关键的。

2.4.2 ICFTC 故障模型

ICFTC 故障模型为：

①ICFTC 设计为容忍单故障，即在故障恢复以前只能容忍系统发生一个故障[2]；

②各复制节点故障发送相互独立；

③系统容忍两种类型的故障：瞬时故障和永久故障。当在固定的时间内发生瞬时故障的次数超过某一设定的门槛时，就认为系统发生了永久故障。

2.4.3 应用进程数据的同步、表决和错误检测

当 PE_i 执行中遇到 OVP 时，它就将与该 OVP 相关数据发送到 $IFMM_i$ 中的内存共享区，进程被挂起。$IFMM_i$ 设定一个时间片，只要在该时间片内收到大于等于 2 份 OVP 数据时进行比较表决(少于 2 份就表示系统发生了某个故障)。通过比较表决检测错误。表决完成后通过中断或者信号机制唤醒被挂起的进程继续执行。图 4 针对一个进程的情况给出了一个简化的同步、表决和错误检测机制。

```
;;====Host side Ci:======
    Start;
    P^a_ij
    {
    ....;
    OUT^0_ij
    {Send results to IFMMi
    Wait();
    If permitted Send out results
    Else disable sending;}
    continue;
            }

    P_ik(k≠j)
    {
    Received notice from IFMMi;
    Wake up P^a_ij
    }
    .....

    ;;====IFMM side:====
    Start: ;
    Self-test;


    If Data from PEj arrived
    {set timer window=T1 and start timer}
    ;
    if timer>T1 AND copy of data <3 or 2
    {Detecting errors;
      Signifying error information;
      Handling error}
    Else If copy of data ==3 or 2 AND timer≤T1
    {Synchronization point;
    Voting and detecting error;
      If no error then notice PEi
      If error then signaling error and
      Start error handling
    Goto STRAT:;
    .....
```

图 4 应用进程数据的同步、表决和错误检测

对 IC 和 CP 可以采用类似的方法，但是 CP 的检测是后台执行的，它由 IFMM 定期对 CP 进行检查。

从图 4 中我们看出：

应用处理与容错处理的并行性：由于有容错用专用处理器，PE_i 的应用处理与 $IFMM_i$ 的容错处理并行执行。与传统的容错处理与应用处理在同一处理器执行不同，这样可以明显减少容错引起的开销；

灵活的容错机制：容错处理如同步、表决、故障/错误检测等在单独的处理器上执行，因此可以根据不同的容错需求设计不同的容错方案，可以满足从最严格的故障检测到部分故障检测甚至是没有故障检测的容错需求。

2.4.4 *多层的故障/错误处理* 上一节中描述的故障/错误处理机制实际上是用来处理逃出系统本身固有的故障/错误处理机制的故障或者错误的。ICFTC 采用多层防护的方法来保证系统的完整性。最底层是固有的硬件故障/错误处理机制。例如，在选用的 PE 和 IFMM 的内存子模块设计中内存总线采用 ECC 编码，它可以纠单错/检双错（Single Error Corrected/Double Error Detected），发生在内存子系统中的任何单故障都可以容忍。CPU 总线包括数据总线、地址总线和命令总线有奇偶校验，可以检测出任一位的单故障，通过指令级的 retry 可以消除 CPU 总线上的任何瞬时单故障。CPCI 总线类似于 CPU 总线的处理。第 2 层是节点内模块间的故障/错误检测处理。如 PE 和 IFMM 就可以互相检测彼此的工作状况。PE 可以检测 IOM 子系统的某些故障。第 3 层是增强的设备驱动程序。使用软件机制（如可接受性测试等）加强对设备异常检测和处理。第 4 层即为 2.4.3 节中描述的机制。通过多层防护，ICFTC 可以达到较高的故障覆盖率。

2.4.4 *透明性实现* 透明性是影响容错计算机使用的一个非常重要的因素。ICFTC 的容错机制是对用户透明的。当应用进程需要输出结果和接受输入时，LINUX 操作系统分别用 write() 和 read() 两个系统调用来处理。通过对这两个系统调用进行改造辅助达到输入一致性和输出的表决。对内存关键区的检测则完全由 IFMM 后台实现。

3 ICFTC 故障覆盖率（Fault tolerance coverage）分析以及容错开销估计

3.1 故障逃逸模型与故障覆盖率分析

如前所述，ICFTC 充分挖掘系统本身固有的故障/错误处理机制，采用多层防护措施来提高系统对故障的容忍能力。发生在低层的故障首先被该层的故障/处理机制检测和恢复，不能被检测和/或恢复的故障就会被上一层的故障/错误处理机制来处理，以此类推。图 5 是 ICFTC 故障逃逸模型。

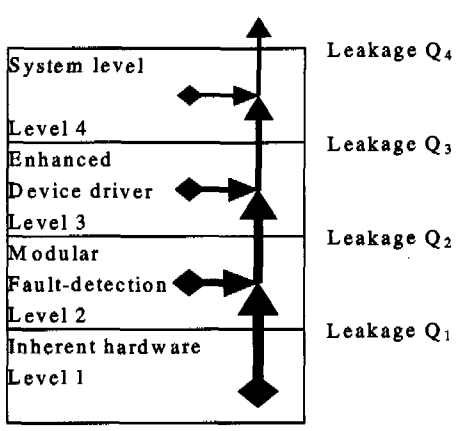


图 5 ICFTC 故障逃逸模型

图中 Q_i 为第 i 层的故障逃逸概率。箭头后部的菱形框表示可能的故障源。箭头表示故障逃逸路径。作如下定义：

记 F_i 为发生在第 i 层的故障；D_i 为被第 i 层检测到的故障；R_i 为被第 i 层恢复的故障，当然肯定是被检测到的故障的一部分；E_i 为逃出第 i 层的故障，包括未被第 i 层检测到的故障和已经被检测到但未能被本层恢复的故障，因此对于第 1 层：

$$E_1=F_1-D_1\cap R_1=(F_1\cap\sim D_1)\cup(F_1\cap D_1\cap\sim R_1)$$

$$Q_1=P(E_1)=(P(\sim D_1)+P(D_1\cap\sim R_1))P(F_1)$$

定义 $C_1=P(R_1)=P(D_1\cap R_1)=(1-(P(\sim D_1)+P(D_1\cap\sim R_1))$ 为第 1 层的故障覆盖率，则：

$$Q_1=(1-C_1)P(F_1) \tag{1}$$

对于第 2 层及后面的层，情况有所不同：该层的故障包括本层发生的故障和从低一层逃逸的故障；本层发生的故障的分析类似于第 1 层的分析，定义本层自覆盖率（$C_i, i=2,3,4$）为发生在本层的故障能被本层恢复的概率，定义逃逸覆盖率为从下一层逃逸到本层且被本层恢复的概率（$K_i, i=2,3,4$）。

所以：

$$P(Q_2)=(1-K_2)P(Q_1)+(1-C_2)P(F_2) \tag{2}$$

$$P(Q_3)=(1-K_3)P(Q_2)+(1-C_3)P(F_3) \tag{3}$$

$$P(Q_4)=(1-K_4)P(Q_3)+(1-C_4)P(F_4) \tag{4}$$

分析：

① 对给定的系统 ICFTC，一旦设计完成后，其每一层的故障发送的概率即故障率一定，并且层数越高，其构造越简单，发生故障的概率也就越低，所以可以假定：

$$P(F_1)\gg P(F_2)\gg P(F_3)\gg P(F_4) \tag{5}$$

② 层的概念是一种抽象的概念，是对每一层容错机制的抽象。

③ 要减少故障逃逸率或者提高整个系统的故障覆盖率，则需要每一层都提高其本层覆盖率和对底层故障的覆盖率。

④ 与传统的采用一层防护的高可靠容错计算机设计相比，采用多层防护的方法可以在每层的故障覆盖率相对较低的情况下获得整体的故障覆盖率与之相当。

对采用单层防护来说：

$$Q_1=(1-C_t)P(F_t) \tag{6}$$

要使(4)和(6)相等，则：

$$(1-K_4)P(Q_3)+(1-C_4)P(F_4)=(1-C_t)P(F_t)$$

进一步：

$$(1-C_t)P(F_t)=(1-K_4)(1-K_3)(1-K_2)(1-C_1)P(F_1)+(1-K_4)(1-K_3)(1-C_2)P(F_2)+(1-K_4)(1-C_3)P(F_3)+(1-C_4)P(F_4) \tag{7}$$

进一步假定 $K_2=K_3=K_4=C_1$，且忽略(7)式的后 3 项（根据(5)的假设），每一层的故障率为负指数分布：$P(F_1)=1-e^{-\lambda_1 \tau}$，$P(F_t)=1-e^{-\lambda_t \tau}$

则：

$$1-C_1=[(1-C_t)(1-e^{-\lambda_t \tau})/(1-e^{-\lambda_1 \tau})]^{1/4} \tag{8}$$

对(8)式，如果假定 $\lambda_1=\lambda_t$，且 $C_t=0.9999$，则 $C_1=0.9$；如果设 $\lambda_1=100\lambda_t=1e^{-4}$/年，则 $C_1=0.9684$。

⑤较高的故障覆盖率意味着更复杂的设计，而更复杂的设计意味着更多的设计缺陷，可靠性就会受到更多的影响。因此采用 COTS 技术和多层故障/错误处理是一种比较合理的、性价比好的设计。

3.2 容错开销估计

T_v 是为检测故障引入表决机制引起的不可避免的时间开销。设 C_i 形成表决数据的时间为 T_1，表决数据在 PHN 的传输时间为 T_2，表决算法的执行时间为 T_3，则 $T_v=T_1+T_2$

（下转封三）

的其他一些未知指令占用。

同样的，考察表1(b)PAPI_CA_SHR的情况，main函数中fac.0的共享Cache线独占申请数占87.64%，fac函数本身中fac.0(递归调用)占88.5%。这样，我们有理由相信，在本实验例子中，fac函数是该程序的主要性能瓶颈，如果需要对程序优化，则主要应该考虑对fac的递归的优化。

结论与展望 本文以一个简单的实例介绍了动态指令编译技术，并结合PAPI技术，建立了Dyninst-PAPI模型，实现了Linux平台中对运行时程序的性能数据采集和分析。通过实验证明，这种技术可以比较有效地对无源代码，或者不能随意重新编译、重新启动的程序进行性能数据采集和分析。所不足的是，目前的实现是在字符控制台中，通过对本地进程的接管和结构分析(路径和关键点的选择还主要依赖分析人员对程序的熟悉程度)，插入监控代码并采集和分析，使得Dyninst-PAPI模型的应用存在较大局限性。今后的研究中，将考虑实现对远程进程的监控以及良好的用户使用接口。如何能够在得到进程的运行镜像后，分析进程的关键路径，自动决策插入监控点位置是一个很有意义和挑战性的研究。在文[8]中介绍了通过分析并发进程间的循环层次结构得到程序的关键路径的方法，但是它的分析主要基于对静态程序结构和数据的分析；同时，通过程序结构分析，得出图形化的程序关键路径表达，提供给普通程序开发和维护人员使用，也是一个很有意义的研究工作。

参考文献

1 Zhang Charles, Jacobsen HansArno. TinyC2: Towards building a dynamic weaving aspect language for C[C]. Foundation of Aspect Oriented Languages Workshop in conjunction with 2nd AOSD Conference 2003, Boston, MA

2 Buck B, Hollingsworth J K. An API for Runtime Code Patching [J]. Jour. of High Performance Computing Applications, Winter 2000, 14(4): 317～329

3 Hollingsworth J K, Miller B P, Cargille J. Dynamic Program Instrumentation for Scalable Performance Tools [C]. 1994 Scalable High-Performance Computing Conf., Knoxville, Tenn. 1994. 841～850

4 Mucci P. DynaProf and PAPI: An Object Code Instrumentation System for Dynamic Profiling [C]. Linux Supercluster Users Conference, 2000

5 ICL of University of Tennessee, PAPI Programmer's Reference [EB/OL], http://icl.cs.utk.edu/

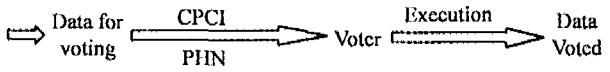
6 Browne S, Dongarra J, Garner N, London K, Mucci P. A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters [C]. In: Proceedings of the IEEE/ACM SC2000 Conference November 04 - 10, 2000 Dallas, Texas

7 Mucci P. PAPI and Dynamic Performance Analysis Technology [R]. report at university of Tennessee, Feb. 2003

8 Dey S, Bommu S. Performance analysis of a system of communicating system [C]. C&C Research Laboratgories, IEEE, 1997

(上接第287页)

$+T_3$，表决流程如下：



可以对本结点机的表决时间做一个基本的估计：

T_1 的时间视表决数据的大小而定，一般为μs级；

CPCI的速度为132MB/S或者512MB/S，PHN的速度为1000Mbps且无阻塞，通过这两个路径几百字节的数据到达表决器的总的时间 T_2 为μs级；

表决执行时间 T_3 与表决算法及同步有关，也为μs级；

所以 T_v 为几十μs到几百μs之间。

由于表决执行与对请求的处理分别在两个处理器上并行执行，因此真正的表决时间可以更小。

进一步工作及结论 ICFTC已经完成研制，在作最后的验证测试。本容错计算机建立在COTS硬件和软件基础上。通过设计一种智能容错管理模块，解决了COTS硬件的可观察性问题，进而实现了将应用任务与容错任务并行处理。在提高容错计算机故障覆盖率方面，ICFTC采用4层故障/错误处理机制，通过对该机制的故障逃逸模型的分析，可以看出ICFTC以较小的设计复杂性和设计开销获得传统定制容错计算机能达到的故障覆盖率。由于容错机制的相对独立性实现，ICFTC结构灵活，具有较好的可扩展性，能满足不同的系统容错要求。

参考文献

1 Yuan Youguang. The Reliability Techniques in Real-Time Systems (in Chinese), Beijing: Tsinghua University Press, Sep. 1995

2 Siewiorek D P, Swarz R S. The Theory and Practice of Reliable System Design: Digital Press, USA, 1982

3 Alkalai L, Tai A T. Long-Life Deep-Space Applications. IEEE Computer, 1998, 31: 37～38

4 Tai A T, et al. COTS-Based Fault Tolerance in Deep Space: Qualitative and Quantitative Analyses of A Bus Network Architecture, In: Proc. of the 4th IEEE Intl. Symposium on High Assurance System Engineering, Nov. 1999. 97～104

5 Tai A T, et al. On-Board Maintenance for Long-Life Systems. In: Proc. of the IEEE Workshop on Application-specific Software Engineering and Technology (ASSET'98). Apr. 1998. 69～74

6 Powell D. A Generic Fault-Tolerant Architecture for Real-Time Dependable Systems, London: Kluwer Academic Publishers, 2001

7 Arlat J. Preliminary Definition of the GUARDS validation strategy: [ESPRIT Project 20716 GUARDS Report]. LAAS_CNRS, FRANCE, Jan. 1997

8 Avresky D R. Dependable Network Computing. Kluwer Academic Publishers, USA, 2000. 3～19

9 Shanley T, et al. PCI System Architecture (fourth edition). Adison Wesley Longman, inc, USA, 1999

10 Ou Zhonghong, et al. A Kind of Generic Real-time Dependable Server Architecture With Low Fault-latency Using COTS Components. In: the Proc. of DCABES'2004, Wuhan, Sept. 2004. 103～109